Non-contiguous Processor Allocation Algorithms for Distributed Memory Multicomputers [*]

Wanqian Liu, Virginia Lo,
and Kurt Windisch

Dept. of Computer and Information Science
University of Oregon
Eugene, OR 97403
Email: wliu, lo, kurtw@cs.uoregon.edu

Bill Nitzberg [†]

NAS Systems Division
NASA Ames Research Center
Moffett Field, CA 94035
Email: nitzberg@nas.nasa.gov

Abstract

Current processor allocation techniques for highly parallel systems have thus far been restricted to contiguous allocation strategies for which performance suffers significantly due to the inherent problem of fragmentation. We are investigating processor allocation algorithms which lift the restriction on contiguity of processors in order to address the problem of fragmentation. Three non-contiguous processor allocation strategies: Naive, Random and the Multiple Buddy Strategy (MBS) are proposed and studied in this paper. Simulations compare the performance of the non-contiguous strategies with that of several well-known contiguous algorithms. We show that non-contiguous allocation algorithms perform better overall than the contiguous ones, even when message-passing contention is considered. We also present the results of experiments on an Intel Paragon XP/S-15 with 208 nodes that show non-contiguous allocation is feasible with current technologies.

1 Introduction

Highly parallel systems have the promise of outperforming traditional vector supercomputers in terms of price/performance on a wide range of individual applications. However, mainstream computing does not run "individual applications," but instead, supports a workload that is a diverse mix of large and small jobs. With respect to overall system utilization, traditional supercomputers are far ahead of parallel systems, the former usually achieving 98-99% utilization. A major price/performance obstacle to highly parallel systems is efficiently supporting workloads.

The processor allocation problem involves the design of algorithms for allocating a set of processors to a given parallel job with the goal of maximizing throughput over a stream of many jobs. Allocation techniques used in current commercial parallel machines, as well as in the research community, have thus far been restricted to **contiguous allocation** in which the processors are constrained to be physically adjacent. In addition, many systems also require that the allocated processors form a subgraph of the original architecture, specifically, subcube allocation in hypercubes and submesh allocation in meshes. The recurring theme found in all these studies is that performance suffers significantly due to internal and external fragmentation, or due to high overheads for allocation and deallocation. **Internal fragmentation** occurs when more processors are allocated to a job than it requests. **External fragmentation** exists when a sufficient number of processors are available to satisfy a request, but they cannot be allocated contiguously. Experimental evidence has shown that little improvement in performance can be realized by refinements of contiguous allocation algorithms [5]. As a result, recent research efforts have focused on the choice of scheduling policies and their impact on contiguous allocation schemes.

Our research takes a different approach to overcoming the limitations of contiguous allocation. We are investigating processor allocation algorithms which lift the restriction on contiguity of processors in order to

---

[*] This research is sponsored by NSF grant MIP-9108528 and the Oregon Advanced Computing Institute

[†] Computer Sciences Corporation, NASA contract NAS 2-12961, Moffett Field, CA 94035-1000

1063-9535/94 $4.00 © 1994 IEEE

address the problem of fragmentation. As we shall show, **non-contiguous allocation** offers several significant advantages over contiguous schemes: elimination of internal and external fragmentation; low allocation and deallocation overheads; compatibility with adaptive processor allocation schemes [10] in which a job may increase or decrease its allocation at runtime; and straightforward extensions for fault tolerance.

Current communication technologies like wormhole routing enable us to consider non-contiguous allocation, since the delay due to the number of hops between processors is known to be negligible. However, we also note that non-contiguous allocation introduces potential problems due to message contention because the messages occupy more links, yielding potential communication interference with other jobs. Therefore, the most successful allocation scheme may be a hybrid between contiguous and non-contiguous approaches.

We compare the performance of three non-contiguous processor allocation strategies: Naive, Multiple Buddy Strategy (MBS), and Random allocation, with three well-known contiguous allocation schemes: Frame Sliding, First Fit, and Best Fit. The strategies we present represent a continuum with respect to degree of contiguity. These strategies are also directly applicable to processor allocation in k-ary n-cubes which include the hypercube and torus.

Section 2 gives a brief summary of previous work in the area of processor allocation for mesh topologies. Section 3 presents the results of preliminary experiments on an Intel Paragon XP/S-15 with 208 compute nodes. Section 4 describes the three non-contiguous schemes and discusses the Multiple Buddy Strategy, an algorithm we have developed that exploits the advantages of non-contiguity with respect to fragmentation while addressing the potential contention that may be introduced. Section 5 analyzes the performance of these strategies through simulation results. Section 6 summarizes our results and discusses future work.

2 Previous Research Work

The Multiple Buddy Strategy proposed in this paper is an extension of the 2-D Buddy Strategy. Our simulations compare the performance of MBS with Frame Sliding, First Fit and Best Fit. The Kreuger paper [5] describes the performance limitations of all contiguous allocation schemes and thus motivates our investigation of non-contiguous approaches.

The *two-dimensional buddy strategy*, a generalization of the one-dimensional binary buddy system for memory management, is proposed by Li and Cheng [6] for a mesh connected system. Under this strategy, all incoming jobs are given square submeshes of size $n' \times n'$ and the system itself is a square mesh of size $n \times n$, where both n' and n are exact powers of 2. The allocation and deallocation overheads of this strategy are all $O(\log n)$, which is relatively low compared to other strategies. However, it can only be applied to square meshes, it suffers from severe internal fragmentation, because a square submesh with side length of 2^i is always required, and it has significant external fragmentation. The Intel Paragon uses an extension to the 2-D buddy strategy which is applicable to non-square meshes and allows allocation across more than one size buddy. [9]

Chuang and Tzeng proposed an improved strategy called the *frame sliding* strategy [3]. It is applicable to any mesh system and any shape of submesh request, thus it has no internal fragmentation. The frame sliding strategy examines the first candidate "frame" from the lowest leftmost available processor and slides the candidate frame horizontally or vertically by the stride of width or height of the requested submesh, respectively, until an available frame is found, or all candidate frames are checked. This strategy has better performance than the 2-D buddy strategy. However, it has higher allocation overhead $O(n)$, it suffers from large external fragmentation, and it cannot recognize all possible free submeshes.

In [13], Zhu proposed the *first fit* and *best fit* strategies, which can be applied for contiguous submesh requests of arbitrary sizes and have the ability to recognize all free submeshes in a system. These algorithms locate submeshes by constructing bit arrays indicating which processors have enough free neighbors to host the *base* node, the lower-left processor. These bit arrays can then be searched for the first available submesh (first fit) or submesh that best fits the request (best fit). Both strategies suffer from significant external fragmentation. These algorithms both have allocation and deallocation overhead of $O(n)$.

Krueger et. al have shown in [5] that increasingly sophisticated processor allocation algorithms do not significantly influence the performance of hypercube systems. Their simulations of four well-known hypercube allocation strategies realized limited improvements despite the differing abilities of these algorithms to reduce fragmentation and recognize available subcubes. The barriers observed by Krueger et. al. are primarily a direct result of external fragmentation, which arises from the contiguity constraint. Although no statistics have been compiled for mesh systems, we believe the same trend will be exhibited under the assumption of contiguity. Thus, improved performance requires exploration of other alternatives, including scheduling policies [2] [8] [11] and the approach we propose: non-contiguous allocation.

3 Contention on Real Systems

The potential increased message contention caused by non-contiguous allocation could greatly reduce performance. As a first step in evaluating the feasibility of non-contiguous allocation, we measured worst-case contention and ran benchmarks on the Intel Paragon XP/S-15 at the Numerical Aerodynamic Simulation (NAS) facility at NASA Ames Research Center. The NAS Paragon is a distributed memory multicomputer with 208 compute nodes connected by a 175 megabyte per second bi-directional mesh, with wormhole, XY routing. In addition to using the operating system supplied by Intel, Paragon OS release 1.1, we ran worst-case contention tests under SUNMOS, a minimal operating system developed by Sandia National Labs and the University of New Mexico.

One would expect message contention to have a noticeable impact on performance; however, we were unable to measure any performance degradation on real applications due to contention. We ran two, four, and six copies of selected NAS Parallel Benchmarks [1] simultaneously and found no measurable performance difference compared with running the benchmarks one at a time on a dedicated system. For example, we partitioned the compute nodes based on diagonals, splitting the machine into four partitions (64, 64, 32, and 32 nodes) in a checkerboard pattern—this pattern was designed to encourage message contention between applications. We ran the best Intel-supplied implementations of selected NAS Parallel Benchmarks (FFT, MG, and CG) in a loop on each partition. The largest performance variation was for FFT, which varied by less than 2% over 13 runs. We saw similar results for other partitioning schemes.

In order to better quantify the effects of message contention on the Paragon, we developed a simple worst-case contention generating program, **contend**. To force contention on the XY routed mesh of the Paragon, we allocated the nodes on the north and east edges of the mesh. Nodes were paired from the middle outward, and each pair exchanged messages. With this configuration, all messages must traverse one common network link. We ran **contend** on up to nine pairs of simultaneously communicating nodes, with message sizes ranging from 0 to 64 kilobytes.

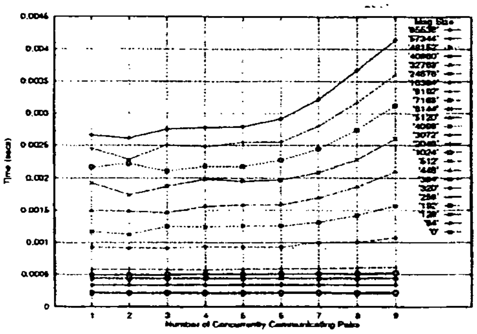

Figure 1: Worst Case Contention on the Intel Paragon (Paragon OS R1.1)

Under the native Paragon operating system (Figure 1), virtually no contention is noticeable (RPC times are flat) through six pairs of communicating nodes. Starting with seven pairs, contention begins to slow message-passing performance, but only for messages larger than 16 kilobytes. This surprising result is an artifact of the current release of the operating system, and explains our inability to measure contention while running multiple simultaneous copies of the NAS benchmarks. Although the Paragon hardware supports 175 megabytes per second bandwidth, the current release of the operating system (R1.1) delivers only about 30 megabytes per second. The hardware has more than enough excess bandwidth to support about six pairs of communicating nodes without any noticeable contention (6 × 30 = 180).

We then ran **contend** on the Paragon under the SUNMOS operating system (Figure 2), which delivers 170 megabytes per second bandwidth, nearly peak speed. With the anomalous operating system behavior eliminated, the effects of contention are significant with only two pairs of communicating nodes, and increase linearly with the number of pairs. However, small messages (less than one kilobyte) appear to be little effected by contention, even with nine pairs of communicating nodes.

Although preliminary, and by no means exhaustive, our experiments show two interesting facts about

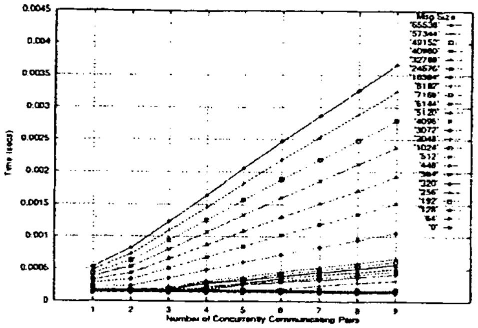

Figure 2: Worst Case Contention on the Intel Paragon (SUNMOS S1.0.94)

contention on the Paragon: current operating system overhead subsumes contention effects, and small messages do not cause contention. The poor operating system performance will likely be corrected in future releases of the Paragon OS, but the contention effects of small messages will likely remain. VanVoorst, et. al. [12], measured the workload of the Intel iPSC/860 system at NAS for ten days, and found that 87% of all messages are, in fact, one kilobyte or less. So, at least for a class of scientific applications, large messages may not be a significant issue. This empirical data is encouraging, and supports the notion that a purely non-contiguous allocation strategy may run into contention effects with large messages, but a purely contiguous strategy is also unnecessary.

4 Non-contiguous Allocation

4.1 Random and Naive Strategies

One of the most straightforward non-contiguous allocation strategies is **Random allocation strategy**, under which a request for k processors is satisfied with k randomly selected processors. Both internal fragmentation and external fragmentation are eliminated, since all jobs are assigned exactly the requested number of processors if available. No contiguity is enforced under this strategy. Another simple non-contiguous allocation strategy is **Naive allocation strategy**, under which a request for k processors is satisfied by the first k free processors in a row major scan of the mesh. Some degree of contiguity is maintained through the nature of the row major scan. Similar to the Random allocation strategy, there will be neither internal nor external fragmentation. The complexity of the allocation and deallocation algorithms for both of these strategies is $O(k)$.

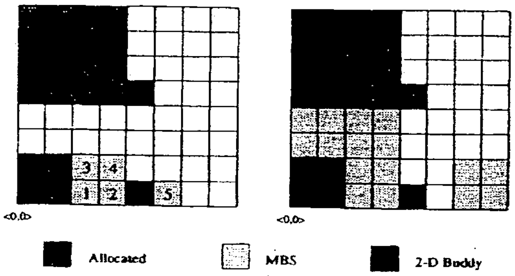

Figure 3: Eliminating system fragmentation using MBS

4.2 Multiple Buddy Strategy

In the following, we introduce another non-contiguous allocation strategy, which we call the *Multiple Buddy Strategy* (MBS). It is an extension of the 2-D buddy strategy[6], which has both internal and external fragmentation problems. MBS eliminates fragmentation by applying the non-contiguous model to the mesh system, while still maintaining contiguity within individual blocks. A more detailed and formal discussion of MBS may be found in [7].

The following scenarios show the problems exhibited by the 2-D buddy strategy and how they are resolved by the MBS strategy. Square submeshes are represented by $< x, y, s >$, in which $< x, y >$ is the location of the lower leftmost processor, and s is the side length of the submesh of size $s \times s$.

In Fig 3(a), a mesh of size $2^3 \times 2^3$ has three allocated submeshes (represented by the black squares): $< 0, 0, 2 >$, $< 4, 0, 1 >$, and $< 4, 4, 1 >$. Assume that a job which needs 5 processors is submitted to the system. Since all submeshes have to be $2^i \times 2^i$ under the 2-D buddy strategy, a 4×4 submesh ($< 0, 4, 4 >$) will be allocated. In this case 11 processors in the submesh will be wasted during the lifetime of the job. Under the MBS strategy, the exact number of processors will be assigned to the job. In the above case, two *blocks* will be assigned to the job: $< 2, 0, 2 >$ and $< 5, 0, 1 >$. Therefore, the internal fragmentation is eliminated.

Assume that the mesh shown in Fig 3(b) receives a request for 16 processors. Since a 4×4 block cannot be found in the mesh using the 2-D buddy strategy, the request will be put into a waiting queue, resulting in external fragmentation. The MBS strategy resolves this problem by breaking a large request into smaller blocks. In the above case, 4 blocks of size 2×2 will be assigned to the job. Since larger requests can always be broken down to 1×1 blocks, there will not be any external fragmentation.

Under MBS a request for k processors is represented as a base 4 number of the form $k = d_m \times 2^m \times 2^m + ... + d_0 \times 2^0 \times 2^0$. MBS attempts to satisfy this request using blocks of sizes $2^m \times 2^m, \cdots, 2^0 \times 2^0$. If a block of a desired size is unavailable, MBS searches for a bigger block which it repeatedly breaks down into buddies until it produces blocks of the desired size. If that fails, MBS breaks a request for a block of size $2^i \times 2^i$ into four requests for blocks of sizes $2^{i-1} \times 2^{i-1}$. This process continues until the original request for k blocks is satisfied. The proposed MBS strategy can eliminate the fragmentation problems and be implemented efficiently. It is composed of the following 5 parts: system initialization, request factoring algorithm, buddy generating algorithm, allocation algorithm, deallocation algorithm.

4.2.1 System Initialization

System initialization is done only once at system startup time. At this time, the whole mesh system is divided into *initial blocks*, which are non-overlapped square submeshes with side lengths that are exactly powers of 2. The initialization process allows the strategy to be applicable to any size mesh system.

A **block** and its **buddies** are defined recursively as follows. Any initial block is a **block**. Each **block** is a square mesh and represented by $< x, y, p >$, where $< x, y >$ is the location of the lower leftmost processor, and p is the side length of the submesh. If $< x, y, p >$ is a **block** and $p > 1$, then $< x, y, \frac{p}{2} >$, $< x+\frac{p}{2}, y, \frac{p}{2} >$, $< x, y+\frac{p}{2}, \frac{p}{2} >$, and $< x+\frac{p}{2}, y+\frac{p}{2}, \frac{p}{2} >$ are **blocks**, and they are **buddies** of each other.

The concept of *free block records* (FBR) extends the notion of the free block lists in the 2-D buddy strategy. $FBR[i]$ records the number ($FBR[i].block\_num$) of available blocks of size $2^i \times 2^i$ and an ordered list ($FBR[i].block\_list$) of the locations of such blocks. At startup time, information about the initial blocks will be kept in $FBRs$, where $FBR[i].block\_num$ keeps the number of $2^i \times 2^i$ initial blocks and $FBR[i].block\_list$ keeps a location list of such initial blocks. Another global variable $AVAIL$, the current number of available processors in the system, is initialized to the number of processors in the mesh system during the startup.

4.2.2 Request Factoring Algorithm

Any integer has a base 4 representation, expressible as a sum $\sum_{i=0}^{\lfloor \log_4 n \rfloor} d_i \times (2^i \times 2^i)$ where $0 \leq d_i \leq 3$. Thus any legal job request can be accommodated by d_i blocks of size $2^i \times 2^i$. At most $\lceil \log_4 n \rceil$ distinct blocks are needed with a maximum of 3 blocks of a given size.

We define the *maximum distinct blocks* (MaxDB) of a given mesh system as $\lceil \log_4 n \rceil$, where n is the number of processors in the system. The factoring algorithm needs to take as input the job size and produces as output a request array ($Request\_Array[0..MaxDB]$), where $Request\_Array[i]$ stands for the number of size $2^i \times 2^i$ blocks that the job needs. The algorithm essentially is an integer conversion algorithm, where $Request\_Array[i]$ is the ith digit in the base 4 integer representation of the job size.

4.2.3 Buddy Generating Algorithm

The buddy generation algorithm breaks a large block into several smaller blocks to satisfy the $2^i \times 2^i$ requests. It contains two phases. In the first phase, an available block is sought by examining the FBRs in increasing order of block size from $2^{i+1} \times 2^{i+1}$ to $2^{max} \times 2^{max}$. During the second phase, the block is repetitively broken down into smaller buddies until the desired size blocks are found. If no block is found in the search phase, as we shall see, the allocation algorithm will break the request down into smaller requests.

4.2.4 Allocation and Deallocation Algorithm

The allocation algorithm includes two main parts. First, the request is factored and stored in Request_Array[i]. If possible, each request for a block of size i is allocated immediatedly from FBR[i]. Otherwise, an attempt is made to satisfy this request from a larger block by breaking it into smaller buddies. If that fails, the request will be broken down into 4 smaller requests, which are stored in Request_Array[i-1]. By the above algorithm, job requests are satisfied with the exact number of processors, and large requests can be accommodated by available smaller blocks; hence we can conclude that the Multiple

Buddy Strategy suffers from neither internal nor external fragmentation.

The deallocation procedure is essentially the same as that of the 2-D buddy strategy. Instead of returning just one block to the system, the MBS strategy needs to return all blocks owned by the job to the mesh system, and merge the buddies up to restore the larger blocks.

Complexity for the allocation algorithm and the deallocation algorithm in the worst case is $O(n)$. For allocation, the accumulated overhead on generate_buddy is $O(\log n)$ and at most $O(n)$ block entries will be allocated, which has $O(n)$ time overhead. For deallocation, since the maximum number of buddy merges is $\frac{n}{4} + \frac{n}{16} + \cdots + 1 = \frac{1}{3}n = O(n)$, the overhead for deallocation in the worst case will not exceed $O(n)$.

5 Performance Analysis

We conducted two distinct sets of simulation experiments to analyze the performance of non-contiguous allocation strategies compared to contiguous ones: (1) fragmentation experiments and (2) message-passing experiments. Our discrete event simulator was implemented in C using the Rice Parallel Processing Testbed Tools YACSIM, a general simulation library, and NETSIM, a library of network simulation extensions [4].

The fragmentation experiments model the arrival, service, and departure of a stream of jobs in a mesh-connected system using first-come, first-serve scheduling (FCFS). These high-level experiments focus on the effects of system fragmentation (both internal and external). Thus, the overhead of allocation and deallocation is ignored in the simulation, and the message-passing behavior of the algorithms is not modeled.

The message-passing experiments model the same stream of jobs, but at a much finer-grained level. The detailed message-passing behavior in a mesh with wormhole routing is simulated down to the level of individual flits and message-passing buffers. The purpose of these experiments is to carefully examine the message contention introduced by non-contiguity.

5.1 Fragmentation Experiments

The first set of experiments, studying the effects of fragmentation on system utilization and job response time, is modeled after the simulation experiments conducted in previous allocation strategy research [13] [3] [5]. In these experiments, jobs arrive, delay for an amount of time taken from an exponential distribution, and then depart. Message-passing is not modeled.

The contiguous allocation strategies simulated in these experiments are First Fit, Best Fit[13], and Frame Sliding [3]. From the non-contiguous strategies, we only present the results for Multiple Buddy Strategy, which performs identically to Random and Naive with respect to system fragmentation. The job request streams were modeled taking the submesh request sizes from the uniform, exponential, increasing, and decreasing distributions. The independent variable in these experiments was the system load, defined as the ratio of the mean service time to mean interarrival time of jobs. Higher system loads reflect the greater demands when jobs arrive faster than they can be processed. For example, under a system load of 1.0, jobs arrive as fast as the are serviced, on the average, and under a system load of 2.0, jobs arrive twice as fast as they can be serviced. See [7] for more simulation details.

For each job size distribution in these experiments, we measure:

- *Finish Time* - the time required for completion of all the jobs.
- *System Utilization* - the percentage of processors that are utilized over time.
- *Job Response Time* - the time from when a job arrives in the waiting queue until the time it completes.

All simulations model a 32 × 32 mesh and run until 1000 jobs have been completed. Results reported for the fragmentation experiments represent the statistical mean after 24 simulation runs with identical parameters, and given 95% confidence level, mean results have less than 5% error.

Table 1 shows how well the four algorithms handle a system saturated by job requests with job sizes taken from each distribution. Simulation results for a heavy system load of 10.0 are presented since, at this load, the system waiting queue is filled very early in the simulation, allowing each allocation strategy to reach its upper limits of performance.

In all cases, the non-contiguous Multiple Buddy Strategy performs much better than First Fit, Best Fit, and Frame Sliding. With uniform, exponential, and decreasing distributions, simulations using MBS finish at least 57% faster than any of the other algorithms and very dramatic improvements are also made

Algorithm	Job Size Distribution			
	Uniform	Expon.	Incr.[a]	Decr.[b]
	Finish Time (simulation time units)			
MBS	365.32	258.68	753.66	119.89
FF	582.01	429.57	882.94	237.90
BF	573.79	428.72	883.08	231.92
FS	608.02	457.88	885.56	267.40
	System Utilization (percent)			
MBS	72.39	69.36	70.18	77.32
FF	45.96	41.68	60.15	39.15
BF	45.70	41.64	60.30	39.28
FS	43.39	38.47	59.84	34.30

[a] $P_{[1,16]} = 0.2, P_{[17,24]} = 0.2, P_{[25,28]} = 0.2, P_{[29,32]} = 0.4$

[b] $P_{[1,4]} = 0.4, P_{[5,8]} = 0.2, P_{[9,16]} = 0.2, P_{[16,32]} = 0.2$

Table 1: Fragmentation experiment results: Finish Time and System Utilization of each algorithm under different job size distributions for a heavy system load (10.0).

in system utilization. Improvement is less dramatic, though still significant, under the increasing distribution because the large job sizes tend to degrade the system towards the point where it can only service one job at a time.

Figure 4 graphs the *system utilization* for these same algorithms and the uniform job size distribution at varying system loads. It shows that MBS can accommodate a much higher system load before becoming overloaded, and that the system utilization at this point is much higher.

The results for contiguous allocation measured in these experiments are all consistent with those reported by Zhu in [13].

These fragmentation experiments indicate that non-contiguous allocation is far superior to contiguous in terms of its ability to utilize the processors. Because non-contiguous allocation can always allocate a job if there are enough processors available, eliminating external fragmentation, it is shown to achieve higher system utilization. Thus, non-contiguous allocation allows for greater job throughput. However, these results ignore the increased communication contention that may be introduced as a result of non-contiguous allocation. Therefore, in order to validate non-contiguous allocation as a viable strategy, experiments must be performed to evaluate message-passing performance.

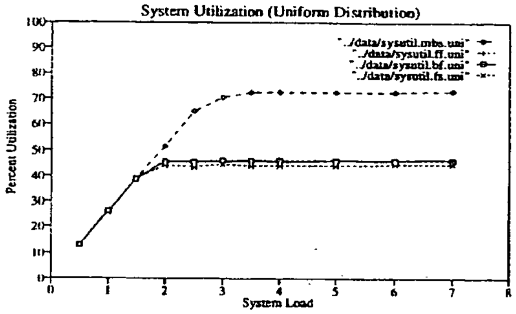

Figure 4: Fragmentation experiment results: System Utilization vs. System Load for the uniform distribution of job size.

5.2 Message-passing Experiments

The second set of experiments measure message-passing contention and its effects on overall performance. The same simulator used in the fragmentation experiments was extended to model the sending and receiving of messages between the processors allocated to a job. Thus, rather than simply delaying for a given service time, processors allocated to the job communicate with each other according to a given communication pattern. The communication pattern iterates until the number of messages sent within the job has reached its message quota, a value taken from an exponential distribution. This quota ensures that the job service times are independent of the job sizes. Once communication ceases, the job departs from the system and is deallocated.

The interconnection network is modeled by XY routing switches. These routing switches are connected by two uni-directional channels to neighboring switches in the mesh and to the corresponding processor elements. The flow control mechanism governing flit movement (flits are the smallest unit of data transmission in the network) is wormhole routing. Messages originate from a processor element and their flits traverse the network in pipeline fashion to their destination processor. If the header flit of a packet is routed to a busy channel, that header flit and its trailing flits stop moving and block whichever channels they occupy in the network. This results in packet blocking time, due to contention, which can be measured in the simulation.

The message-passing experiments implement five communication patterns: all-to-all broadcast, one-to-all broadcast, the n-body computation, fast fourier transform (FFT), and multigrid (MG) from the NAS parallel benchmarks. These cover many communications patterns used very frequently by highly parallel applications and provide a spectrum of message passing complexity ranging from $O(n)$ to $O(n^2)$.

For simplicity and consistency, the internal mapping of the processes within each job is a row-major ordering of processors in each contiguously allocated block. This makes the latter three patterns very interesting cases, since the row-major mapping of these patterns is well-suited to contiguous allocations. These cases will be examined in more detail below.

Using this simulation model, experiments were conducted for each communication pattern using job streams generated from the uniform distribution. The network communication delay parameters were chosen such that the average job service times were great enough to result in high system loads, and thus, minimal system fragmentation. Experimental results are presented for Multiple Buddy Strategy, Random, Naive, and First Fit allocation. First Fit was chosen as a representative of contiguous allocation strategies since it has been shown to perform as well as the others [13]. See [7] for more simulation details.

From each simulation run, we measure:

- *Finish Time* - the time required for completion of all the jobs. Finish time is a good measure of overall performance.
- *Service Time* - the time from when a job begins execution until the time it completes. Service time includes the total communication latency for the communication pattern executed.
- *Packet Blocking Time* - the time that a packet is blocked in the network waiting for a channel to become free. Packet blocking time is a measure of the contention.
- *Weighted Dispersal* - the degree of non-contiguity for an allocation, approximating the percentage of links that are potential sources of contention. Dispersal is defined as the number of unallocated processors divided by the total number of processors in the smallest rectangle circumscribing all processors allocated to a specific job. The weighted dispersal, then, is the job's dispersal multiplied by the number of processors allocated to the job.

All message-passing simulations model a 16×16 mesh and run until 1000 jobs have been completed. Results reported represent the statistical mean after 10 simulation runs with identical parameters and, given 95% confidence level, the mean results have less than 5% error, with the exception of service times, which have less than 10% error.

Table 2(a) shows the results of simulations for jobs executing the heavy All-to-All communication pattern. As expected, contiguous allocation shows the least amount of contention (as seen in the packet blocking times). However, based on the overall finish times, MBS and Naive significantly outperform the other strategies. The reason is that, although they suffer slightly more contention than contiguous allocation, the improvements in system utilization still outweigh the increased communication overhead. It is also interesting to note that MBS and Naive allocation result in only moderate dispersal when compared to Random allocation, which performs poorly.

Table 2(b) shows the results of simulations for jobs executing the One-to-All communication pattern. Under the lighter traffic load induced by this pattern, the contention effects seen in the experiments with All-to-All, are reduced. Again MBS and Naive perform best overall, showing only moderate dispersal and contention. Contiguous allocation finishes last, taking 42% more time than MBS.

Table 2(c) shows the results of simulations for jobs executing the n-body communication pattern. The packet blocking times show that contiguous strategies have very little contention for this pattern, in which almost all communication occurs between adjacent neighbors when mapped by a row-major ordering. For MBS and Naive, contention increases somewhat, but still remains relatively low due to the fact that some degree of contiguity is maintained. This allows the ring communication to still be executed efficiently. The increased contention for MBS and Naive allocation is not significant enough to outweigh the improvements in system utilization. Random performs much worse than any of the others since it cannot take advantage of the regular ring communication in the n-body. Overall, MBS and Naive still finish faster than either Random or contiguous allocation.

Tables 2(d) and 2(e) show the results of simulations for jobs executing two communication patterns that are well matched to the mesh topology of the target machine. Due to restrictions imposed by the communication pattern, all job request sizes were rounded to the nearest power of two in these experiments. Be-

(a) All-To-All Broadcast

Algorithm	Finish Time	Average Packet Blocking Time	Weighted Dispersal
Random	326620	33.968	42.037
MBS	273987	29.216	26.717
Naive	232157	21.990	14.832
First Fit	323343	21.154	0

(b) One-To-All Broadcast

Algorithm	Finish Time	Average Packet Blocking Time	Weighted Dispersal
Random	5454	0.40980	42.298
MBS	5045	0.36506	27.002
Naive	5105	0.36700	14.911
First Fit	7166	0.35001	0

(c) n-Body

Algorithm	Finish Time	Average Packet Blocking Time	Weighted Dispersal
Random	26219	0.228657	41.916
MBS	9044	0.013394	29.956
Naive	8990	0.014407	18.400
First Fit	11903	0.004326	0

(d) 2D FFT

Algorithm	Finish Time	Average Packet Blocking Time	Weighted Dispersal
Random	2431	0.21896	32.302
MBS	968	0.15387	12.161
Naive	1352	0.19339	14.470
First Fit	774	0.07494	0

(e) NAS Multigrid Benchmark

Algorithm	Finish Time	Average Packet Blocking Time	Weighted Dispersal
Random	3132	0.21734	31.826
MBS	1083	0.08051	12.0389
Naive	1841	0.24005	14.298
First Fit	1195	0.09228	0

Table 2: Message-passing experiment results for the five communication patterns.

cause both communication patterns are optimized to perform best in a mesh allocation whose side lengths are powers of two, they perform efficiently with contiguous allocation. However, since MBS allocates multiple such submesh blocks to each job, message passing is also surprisingly efficient under this allocation strategy. Therefore, with these highly mapping-sensitive applications, MBS performs nearly as well or better than the contiguous strategies, and Naive and Random allocation perform very poorly.

From these message-passing experiments, it appears that MBS and Naive allocation strategies outperform both contiguous and Random non-contiguous allocation, with higher system utilization and increased job throughput, reflected in their faster finishing times. They take advantage of the greater flexibility offered by non-contiguous allocation while still maintaining a degree of contiguity, as reflected in their moderate dispersal values. The packet blocking times indicate that this pays off in performance because contention is reduced in comparison to Random allocation. We would expect contention effects to be even less significant in real parallel applications, where only a portion of the total execution time is spent in communication.

6 Conclusions

This paper investigates non-contiguous processor allocation strategies as a method for improving performance in message-passing multicomputers. Contiguous allocation schemes suffer from low utilization due to serious fragmentation problems, and experiments have demonstrated that there is a limit to the amount of improvement that can be achieved for contiguous allocation.

We study three non-contiguous processor allocation strategies for mesh-based multicomputers and compare their performance with that of several well-known contiguous allocation schemes. To summarize our results:

- **Non-contiguous allocation strategies dramatically outperform contiguous allocation strategies with respect to fragmentation.** As a result system utilizations for non-contiguous schemes reach as high as 77% compared to utilizations of 34% to 46% for contiguous schemes when message-passing contention is not considered.
- **The non-contiguous allocation algorithms perform better overall than the contigu-**

ous ones, even when message-passing contention is considered. The increased contention due to non-contiguous allocation is not as serious as the fragmentation effects of contiguous allocation.

- **Non-contiguous allocation strategies that take advantage of non-contiguity while providing some degree of contiguity exhibit the best performance.** The fully contiguous First Fit algorithm and the fully non-contiguous Random allocation algorithm exhibited the worst performance, while the best performance was achieved by the Multiple Buddy Strategy and Naive allocation.

- **Non-contiguous allocation is feasible on present day multicomputers with wormhole routing.** Current operating system overhead on the Paragon XP/S-15 (Paragon OS R1.1) subsumes the contention effects under non-contiguous allocation. Even under improved operating systems, the contention effects are negligible for small messages (less than one kilobyte). A sample workload at NASA NAS shows 87% of all messages to be one kilobyte or less.

Our study shows that non-contiguous strategies yield dramatic improvements in system performance because they eliminate both internal and external fragmentation. Furthermore, the amount of contention introduced by non-contiguity can be limited so their effects on utilization and throughput are minimized. We conclude that non-contiguous allocation provides a new approach that will help highly parallel systems achieve excellent price/performance ratios in a high demand, multi-user environment.

Acknowledgements

Thanks to Yahui Zhu for providing us with his simulator code and to Jayne Miller for carefully reading the manuscript.

References

[1] David H. Bailey, Eric Barszcz, Leonardo Dagum, and Horst D. Simon. NAS Parallel Benchmark Results 3-94. Technical Report Technical Report RNR-94-006, NASA Ames Research Center, Moffett Field, CA 94035-1000, March 1994.

[2] Sourav Bhattacharya, Lionel M. Ni, and Wei-Tek Tsai. Lookahead Processor Allocation in Mesh-Connected Massively Parallel Computers. Technical report, University of Minnesota Dept. of Computer Science, 1993.

[3] Po-Jen Chuang and Nian-Feng Tzeng. An Efficient Submesh Allocation Strategy for Mesh Computer Systems. In *1991 International Conference on Distributed Computer Systems*, pages 256–263, 1991.

[4] R. Covington, S. Dwarkadas, J. Jump, J. Sinclair, and S. Madala. The efficient simulation of parallel computer system. *International Journal in Computer Simulations*, 1:31–58, 1991.

[5] Phillip Krueger, Ten-Hwang Lai, and Vibha A. Dixit-Radiya. Job scheduling is more important than processor allocation for hypercube computers. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):488–497, May 1994.

[6] Keqin Li and Kam-Hoi Cheng. A Two-Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System. *Journal of Parallel and Distributed Computing*, 12:79–83, 1991.

[7] Wanqian Liu, Virginia Lo, and Kurt Windisch. Non-contiguous processor allocation algorithms for distributed memory multicomputers. Technical Report CIS-TR-94-11, University of Oregon, 1994.

[8] Prasant Mohapatra, Chansu Yu, Chita R. Das, and Jong Kim. A lazy scheduling scheme for improving hypercube performance. In *International Conference on Parallel Processing*, pages 110–117, 1993.

[9] Reagon Moore. Intel Paragon Allocation Algorithms. San Diego Supercomputing Center. Personal Communication, 1994.

[10] Vijay K. Naik, Sanjeev K. Setia, and Mark S. Squillante. Performance Analysis of Job Scheduling Policies in Parallel Supercomputing Environments. In *Proceedings Supercomputing '93*, pages 824–833, 1993.

[11] B. Narahari and R. Krishnamurti. Scheduling independent tasks on partitionable hypercube multiprocessors. In *Proceedings of the International Parallel Processing Symposium*, pages 118–122, 1993.

[12] Brian VanVoorst, Steven Seidel, and Eric Barszcz. Profiling the Communication Workload of an iPSC/860. In *Scalable High Performance Computing Conference*, May 1994. to appear.

[13] Yahui Zhu. Efficient Processor Allocation Strategies for Mesh-Connected Parallel Computers. *Journal of Parallel and Distributed Computing*, 16:328–337, 1992.